This is an attempt to have a consistent story around SML references. (sml:ref="true"; not including sml:keyrefs.) This covers SML Bugzillas bugs 4658, 4673, 4682, 4683, 4780/4795, 4834, 4865, 4884, 4976.

Some of concepts and suggestions are already implemented in the first public working draft of the specifications. They are included here to make this proposal complete. Resolutions to Bugzilla bugs are marked with the corresponding bug number.

## 1. What's an SML reference?

An element information item in an SML instance document is an SML reference if and only if it has an attribute information item whose {name} is "ref" and whose {namespace} is <sml namespace> and whose {normalized value}, after whitespace normalization using "collapse" following schema rules, is either "true" or "1".

Note: this doesn't depend on having a schema, and will not be affected if a schema is used to produce a PSVI.

Rationale: schemas may not always exist. People need to understand SML documents with SML semantics (just like people can process XML documents without a schema; perform XSLT without a schema.)

## 2. How are reference elements handled?

If element R is a non-null SML reference, then for reference-handling purposes, R is processed as a unit by SML consumers (who know how to resolve such reference; including their implementations of the deref() function). Note that the above only applies to reference-handling. Often, R may also have non-reference-scheme content and/or attributes (collectively: information items) that may be used in other contexts for other purposes, in which case it doesn't need to be viewed as a unit.

Producers decide which reference scheme(s) to use; in order to be useful, at least one reference scheme used by the producer of an SML-IF document should be understood by each consumer of the document. Each consumer decides which reference scheme(s) to recognize, and this choice MUST be consistent between the consumer's mainline and deref() components, which may be separate implementations.

Every consumer must follow a set of rules when processing a reference, but there is a minor difference between the processing required of validators and that required more generally of all consumers. This happens when a single reference R contains multiple schemes recognized by the consumer or the recognized scheme(s) resolve(s) to more than one target. Since a particular consumer may only use some of the SML-defined semantics, it may choose to short-circuit the reference resolution process after finding one scheme it recognizes that resolves successfully; a validator, since it is expected to implement *all* the SML-defined semantics, must evaluate all reference schemes that it recognizes. It is perfectly acceptable for a general consumer to follow exactly the same rules required for a validator, e.g. to facilitate code re-use between validator and non-validating consumers.

The general form for handling non-null references is shown below.

1. If the consumer recognizes no scheme used in the reference, then R is unresolved
2. Assume the consumer recognizes R as using N schemes supported by the consumer, then (answers bug [4976]):
   2.1. The validator consumer MUST attempt to resolve R using all N schemes, and
      2.1.1. If at least one of the recognized schemes resolves to more than one target element, then report an error.
      2.1.2. If one scheme resolves to a target that's different from the target resolved by another scheme, then report an error.
      2.1.3. If one scheme resolves and another doesn't, then report an error.

2.1.4. If none of the above is true (that is, all recognized schemes resolve to the same one and only one target element, call it T), then R is resolved to T.

2.2. Non-validator consumers are not required to attempt to resolve all N schemes. Their behavior in this case is implementation-defined, but it has to satisfy the following constraints:

2.2.1. It either treats R as unresolved, or resolves R to a single target T.

2.2.2. If none of the recognized schemes resolves, then R is unresolved.

2.2.3. If R is treated as unresolved, then at least one the above 2.1.1~2.1.3 is true for those schemes that are attempted by the consumer.

2.2.4. If R resolves to T, then T MUST be among the target(s) resolved by at least one recognized scheme.

In summary, the consumer has to obey the “at most one target” rule, and it can’t lie about the result. The above rules allow a general consumer to behave in (and not limited to) any of the following ways or any combination of them:

- Only use the first scheme it recognizes, and ignore other schemes.
- Try the recognized schemes one by one until one resolves.
- If a scheme or multiple schemes resolve to more than one target, make R unresolved.
- If a scheme or multiple schemes resolve to more than one target, pick one of them as the target for R.

Note: implementation-defined means a consumer MUST clearly *document* its behavior.

Note: only validators are required to report “multiple target” errors. This means the current “3.1.2.1 At Most One Target” constraint needs to be updated to only cover generic consumers. Validator behavior needs to be moved to a different section.

Any attempt to resolve a reference is limited to the *current* model. That is, target MUST be within the current model.

So any given reference falls into one of 3 categories: null, resolved, and unresolved. (More about “null” in section 4”.) Suggest we delete the definition of “dangling reference” and use “unresolved reference” instead, because the former doesn’t carry more information than the latter. For unresolved references, it’s not always possible to tell whether it’s because the reference is wrongly specified or whether the target is outside of the model. (But “dangling” would imply the latter case.)

Implementations of the deref() function should be governed by the same rules as generic consumers. A deref() implementation should not report errors (which isn’t allowed by XPath 1.0 anyway). For a given reference element, it MAY either return nothing (unresolved) or one target. It MUST NOT return more than 1 target for each reference element. (Answers bug [4683].)

Rationale: it's a unit, because we should not constrain how schemes are defined. e.g. maybe my scheme uses multiple <my:uri> elements, and I specify "the reference resolves to the first <my:uri> that actually resolves". Then even though there could be many <my:uri> sub-elements, they are *not* multiple instances of the same scheme and don't constitute an error if more than one resolves.

There is a question about "when no scheme is recognized, should it be viewed as a null or unresolved reference?" The above gives the answer "unresolved", because this case is distinct from “this reference doesn’t have a value (null)”.

Because all schemes used by the same reference element are meant to target the same element, it should then be an error for cases described in 2.3, which should be reported by validators. But for reasons like performance, general consumers may not want to try to resolve all schemes, hence the implementation-defined behavior.

## 3. How are schemes defined?

Again because references are units, we don't need to get into details in terms of how new schemes should be defined. In particular:

- Schemes may overlap. That is, the same sub-element/attribute can be used by multiple schemes.

- Schemes can use attributes. (More on this follows.)

The only questions a scheme definition needs to answer are: (Answers bug [4865].)

- How is a scheme recognized. That is, there is a set of rules that, when satisfied, identify elements with sml:ref="true" as instances of this scheme. (e.g. for the "uri/iri" scheme, the rule can include: there is one and only one sub-element named <sml:uri>, whose value is xs:anyURI.)
- How is a scheme de-referenced. That is, for an element R using this scheme, a deterministic approach is defined to resolve R to a set of target element nodes.

Because rules are specified in terms of the reference elements, we don't need to deal with "what if there are 2 instances of the same scheme in a single reference R". (Answers bug [4658].)

Rationale: again, we don't want to constrain how people want to define their schemes, especially for documents that pre-date SML. All we can require is that given an SML reference element R (with sml:ref="true"), a scheme must be defined in a way that a) consumers can tell whether R uses such a scheme, and b) if so how to resolve R using the scheme to a set of nodes. (The set may contain more than 1 node, which will result in an error because SML references must resolve to a single node, but no one else cares about why there were more than 1.)

## 4. How are null references identified and handled?

“Null” references are distinct from unresolved references. A null reference is an explicit declaration of intent by the producer that the reference itself does not exist, and a processing directive (NOT a hint) to consumers not to search the reference for reference scheme information items.

Propose to introduce another attribute, say, sml:nilref, with a boolean value (following the same rules as sml:ref). This attribute is only allowed when sml:ref="true"; if sml:nilref 's value is "true", then the reference is viewed as "null". This is the only case where an reference element denotes a “null” reference. (Answers bug [4884].) In the SML schema, define a global sml:nilref attribute with xs:boolean type.

When an reference element is recognized as “null”, then consumers MUST not attempt to resolve it. Any scheme-related content (including attributes) recognized by the consumer is ignored for reference-related purposes. The question of whether a null reference is resolved or not is undefined; it is an ill-formed question.

“targetRequired” should also apply to “null” references. That is, it’s an error if targetRequired=true is specified and the corresponding reference element R has sml:nilref=true. (Answers bug [4780/4795].)

Rationale: Null references are different from empty references (e.g. content of the reference element is intentionally empty; this is an explicit declaration of intent from the producer). Empty references are just normal references (whose value happens to be empty).

Using the university example, if each course has a “primary teacher” and a “assistant teacher”, wouldn’t it be nice for the consumer/application to know “this course doesn’t have an assistant teacher” rather than “I (the consumer) don’t understand the way you (the producer) specified the assistant teacher, and I don’t know who (s)he is”?

As for recognizing “null” references, xsi:nil doesn't work unless we constrain scheme definitions to not use attributes (i.e., to use element content only). Also when schema validation is used, xsi:nil disallows any sub-element or textual content; but there may be cases where a null reference still has non-reference-related content.

This also answers the question "do we support schemes involving attributes"? Because whether a reference is null or not is controlled by sml:nilref, it can cover attributes well (whereas xsi:nil can't). (Answers bug [4682].)

“Null” should not be a special case to “targetRequired”. If we view “nilref” as similar to “xsi:nil” in schema, then “targetRequired=true” is like the opposite of “nillable”.

## *5. What's an SML reference type in schema?*

Note: saying that a schema type is an “SML reference type” implies that instances of the type are expected to hold SML references, and reference-specific constraints can be applied to them.

Options, from least to most flexible:

1. sml:refType and only types derived from it are viewed as SML reference types
2. An XML Schema complex type is an SML reference type if and only if it includes a reference to the sml:ref global attribute declaration in its {attribute uses}, with {required} = true and {value constraint} = fixed + “true”. This says that for an element information item to be valid with respect to an SML reference type, it MUST have sml:ref present with value “true”. That is, it MUST be a reference element.
3. all complex types are SML reference types; whether they really are used as references is determined at runtime by whether the instance has sml:ref="true"

Proposal: option 2 (Answers bug [4673].)

With option 2 adopted, sml:acyclic and sml:targetXXX attributes can only be specified on SML reference types or element declarations whose types are SML reference types. (Answers bug [4834].) If they are used elsewhere, it is an error. In particular, if a complex type has a reference to the global sml:ref attribute declaration but the reference either is optional or doesn’t have a fixed value “true”, then it’s not an SML reference type, and it’s an error to use sml:acyclic or sml:targetXXX with this complex type.

(Similar to schema disallowing defining new xsi: attributes, see [1]) SML can disallow defining sml: attributes (by user schemas). (This removes the possibility for a schema to define a local sml:ref, which would confuse consumers.)

[1] http://www.w3.org/TR/xmlschema-1/#no-xsi

Suggest to remove the special sml:refType (as it has no special meaning any more).

Rationale:

- Advantages of #2/#3 over #1: allows you to use your existing types for references (as schema don't support multiple inheritance or “multi-typing” as John’s email called it)

- Advantages of #1/#2 over #3: makes it possible to identify reference types at compile time (a good property for static schema/grammar analysis)

- Advantages of #1/#2 over #3: removes the possibility where an instance without an sml:ref attribute or with sml:ref=”false” and either acyclic or targetXXX specified. The spec would then have the problem of defining what that means semantically.

- Advantages of #2 over #3: it's very unlikely for a schema author to specify acyclic/targetXXX on a complex type but still not know whether the corresponding instance will be an SML reference.

- Advantages of #1/#2 over #3: being able to know which types are "real" reference types also helps, e.g., GUI-based tooling.